Procedural Stochastic Textures by Tiling and Blending

Thomas Deliot and Eric Heitz

1.1 Introduction

Heitz and Neyret recently introduced a new by-example procedural texturing method for stochastic textures, typically natural textures such as moss, granite, sand, bark, etc. [Heitz and Neyret 18]. Their algorithm takes as input a small texture example and synthesizes an infinite output with the same appearance, as in Figure 1.1. The algorithm is a simple tiling-and-blending scheme augmented by a histogram-preserving blending operator that prevents the visual artifacts caused by linear blending. The cornerstone of the implementation is thus this new blending operator, which requires dedicated precomputations. In this chapter, we investigate the details of a practical implementation of this algorithm with some improvements compared to the original article. The chapter comes with a C++ OpenGL demo the code snippets are extracted from.

Figure 1.1. Procedural stochastic textures by tiling and blending. *Our algorithm runs in a fragment shader that requires no more than 4 texture fetches and a few computations. It can be efficiently integrated into a rendering engine.*

1.2 Tiling and Blending

The fragment shader of our tiling-and-blending algorithm is illustrated in Figure 1.2. We partition the uv-space on a triangle grid and compute the local triangle and the barycentric coordinates inside the triangle. We use a hash function to associate a random offset with each vertex of the triangle grid and use this random offset to fetch the example texture. Finally, we blend the result using the barycentric coordinates as blending weights. This method is fast because each pixel requires only a few computations and 3 texture fetches. The implementation is provided in Listing 1.1.

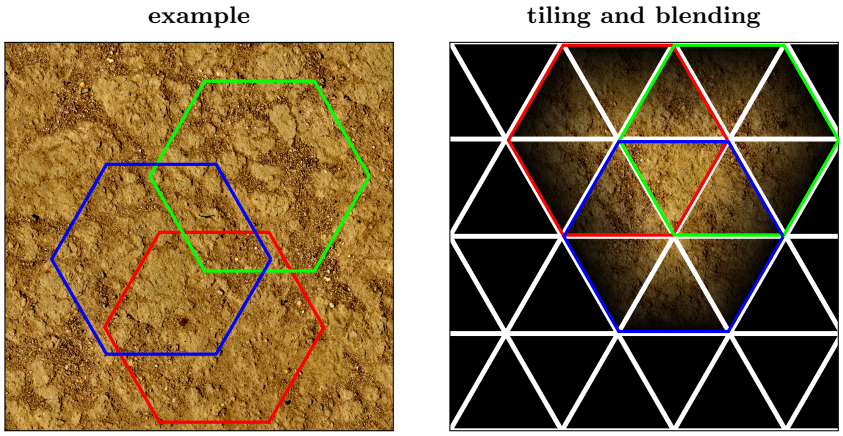

Figure 1.2. Tiling and blending. *Each pixel is obtained by blending three tiles from the example.*

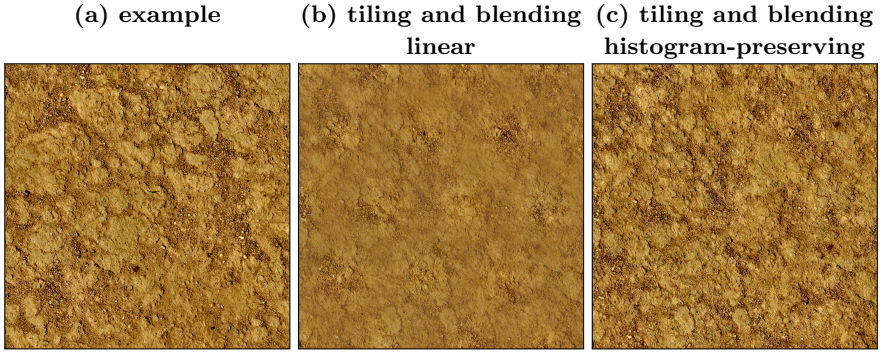

Figure 1.3. Results of tiling and blending. *The tricky part of the algorithm is the blending operator.*

```
sampler2D input; // Example texture

vec3 ProceduralTilingAndBlending(vec2 uv)
{
    // Get triangle info
    float w1, w2, w3;
    ivec2 vertex1, vertex2, vertex3;
    TriangleGrid(uv, w1, w2, w3, vertex1, vertex2, vertex3);

    // Assign random offset to each triangle vertex
    vec2 uv1 = uv + hash(vertex1);
    vec2 uv2 = uv + hash(vertex2);
    vec2 uv3 = uv + hash(vertex3);

    // Precompute UV derivatives
    vec2 duvdx = dFdx(uv);
    vec2 duvdy = dFdy(uv);

    // Fetch input
    vec3 I1 = textureGrad(input, uv1, duvdx, duvdy).rgb;
    vec3 I2 = textureGrad(input, uv2, duvdx, duvdy).rgb;
    vec3 I3 = textureGrad(input, uv3, duvdx, duvdy).rgb;

    // Linear blending
    vec3 color = w1*I1 + w2*I2 + w3*I3;

    return color;
}
```

Listing 1.1. Tiling and blending.

1.2.1 Tiling

In this section, we provide the implementation of the functions required for the tiling part of the algorithm in Listing 1.1.

Triangle grid We use the equilateral-triangle lattice introduced in Simplex Noise [Perlin 01]. Listing 1.2 provides the function that, for a given point in uv space, computes the vertices of its containing triangle and its barycentric coordinates w_1, w_2, w_3 inside this triangle. With this partitioning of the uv space, each vertex is associated with a hexagonal tile chosen randomly in the input image such that each point is covered by exactly 3 tiles and each tile is weighted by a function falling to 0 at the borders and such that the sum of the weights equals 1 everywhere ($w_1 + w_2 + w_3 = 1$). Note that the constant $2\sqrt{3}$ controls the size of the input with respect to the size of the tiles. With this value, the height of a hexagonal tile is half the size of the input texture, which works well in general. This parameter can be adjusted depending on the input. Using larger tiles (decreasing the constant) captures more large-scale features but is more prone to visible repetitions. Using smaller tiles (increasing the constant) increases the variety of the tiles but misses large-scale features.

```
// Compute local triangle barycentric coordinates and vertex IDs
void TriangleGrid(vec2 uv,
    out float w1, out float w2, out float w3,
    out ivec2 vertex1, out ivec2 vertex2, out ivec2 vertex3)
{
    // Scaling of the input
    uv *= 3.464; // 2 * sqrt(3)

    // Skew input space into simplex triangle grid
    const mat2 gridToSkewedGrid = mat2(1.0, 0.0, -0.57735027, ↩
        1.15470054);
    vec2 skewedCoord = gridToSkewedGrid * uv;

    // Compute local triangle vertex IDs and local barycentric ↩
        coordinates
    ivec2 baseId = ivec2(floor(skewedCoord));
    vec3 temp = vec3(fract(skewedCoord), 0);
    temp.z = 1.0 - temp.x - temp.y;
    if (temp.z > 0.0)
    {
        w1 = temp.z;
        w2 = temp.y;
        w3 = temp.x;
        vertex1 = baseId;
        vertex2 = baseId + ivec2(0, 1);
        vertex3 = baseId + ivec2(1, 0);
    }
    else
    {
        w1 = -temp.z;
        w2 = 1.0 - temp.y;
        w3 = 1.0 - temp.x;
        vertex1 = baseId + ivec2(1, 1);
        vertex2 = baseId + ivec2(1, 0);
        vertex3 = baseId + ivec2(0, 1);
    }
}
```

Listing 1.2. Computing the local triangle vertices and barycentric coordinates.

Hash function We use the hash function given in Listing 1.3 to associate a random offset with each vertex of the triangle grid and use it to fetch the example texture. The choice of the hash function does not really matter as long as it provides enough randomness and does not introduce visible correlations between neighboring tiles.

```
vec2 hash(vec2 p)
{
    return fract(sin((p) * mat2(127.1, 311.7, 269.5, 183.3) )↩
        *43758.5453);
}
```

Listing 1.3. The hash function used to randomize the tiles.

Fetching the example texture We fetch the input texture with mipmapping and anisotropic filtering like a conventional texture. Note that the hardware uses screen-space derivatives to compute the mipmap level and parameterize its anisotropic filter. Typically, these derivatives are computed with the finite differences between neighboring pixels of the uv positions passed as argument to the **texture** function. In our case, these screen-space derivatives are broken by the random offsets if neighboring pixels are not in the same triangle. To avoid this problem, in Listing 1.1 we compute the uv derivatives before adding the random offsets and we pass them explicitly to the **texture2DGrad** function.

1.2.2 Blending

In this section, we address the blending part of the algorithm in Listing 1.1.

The problem of linear blending Listing 1.1 implements a classic linear blending operator:

$$I = w_1 I_1 + w_2 I_2 + w_3 I_3. \tag{1.1}$$

Unfortunately, it does not yield satisfying results, as shown in Figure 1.3(b). The result has heterogeneous contrast and exhibits a grid-revealing pattern. Heitz and Neyret explain that the problem of linear blending is that is does not preserve the statistical properties of the input, i.e. its histogram. The problem is thus to find a blending operator that preserves the histogram.

Variance-preserving blending Heitz and Neyret notice that in the special case where the input has a Gaussian histogram, variance-preserving blending preserves the Gaussian histogram. The expression of this operator is

$$G = \frac{w_1 G_1 + w_2 G_2 + w_3 G_3 - \mathbb{E}[G]}{\sqrt{w_1^2 + w_2^2 + w_3^2}} + \mathbb{E}[G], \tag{1.2}$$

where the expectation $\mathbb{E}[G]$ is the average color of the Gaussian input.

Histogram-preserving blending To generalize this idea to arbitrary non-Gaussian inputs, Heitz and Neyret use an histogram transformation T that makes the input Gaussian, blend with the variance-preserving blending of Equation (1.2), and finally apply the inverse histogram transformation T^{-1}. The overview of this algorithm is provided in Figure 1.4. This operator provides better results than linear blending, as shown in Figure 1.3(c). The following is dedicated to the implementation of this operator in the tiling-and-blending algorithm. For more details on histogram-preserving blending, we refer the reader to the original article [Heitz and Neyret 18].

Precomputations The histogram-preserving version of the tiling-and-blending algorithm requires the Gaussian version of the input $T(I)$ and the inverse histogram transformation T^{-1}. We pass them to the fragment shader as textures in Listing 1.4. Section 1.3 is dedicated to the precomputation of these textures.

```
uniform sampler2D Tinput; // Gaussian input T(I)
uniform sampler2D invT; // Inverse histogram transformation T^{-1}
```

Listing 1.4. Textures for histogram-preserving blending.

Fragment shader We update the blending step of Listing 1.1 with the instructions provided in Listing 1.5. Instead of sampling the original input, we sample the Gaussian input stored in texture **Tinput** and we use the variance-preserving blending operator of Equation (1.2). Finally, we apply the inverse histogram transformation by fetching the precomputed look-up table stored in texture **invT**.

```
    // Sample Gaussian values from transformed input
    vec3 G1 = textureGrad(T_input, uv1, duvdx, duvdy).rgb;
    vec3 G2 = textureGrad(T_input, uv2, duvdx, duvdy).rgb;
    vec3 G3 = textureGrad(T_input, uv3, duvdx, duvdy).rgb;

    // Variance-preserving blending
    vec3 G = w1*G1 + w2*G2 + w3*G3;
    G = G - vec3(0.5);
    G = G * inversesqrt(w1*w1 + w2*w2 + w3*w3);
    G = G + vec3(0.5);

    // Fetch LUT
    vec3 color;
    color.r = texture(Tinv, vec2(G.r, 0)).r;
    color.g = texture(Tinv, vec2(G.g, 0)).g;
    color.b = texture(Tinv, vec2(G.b, 0)).b;
}
```

Listing 1.5. Implementation of histogram-preserving blending in Listing 1.1.

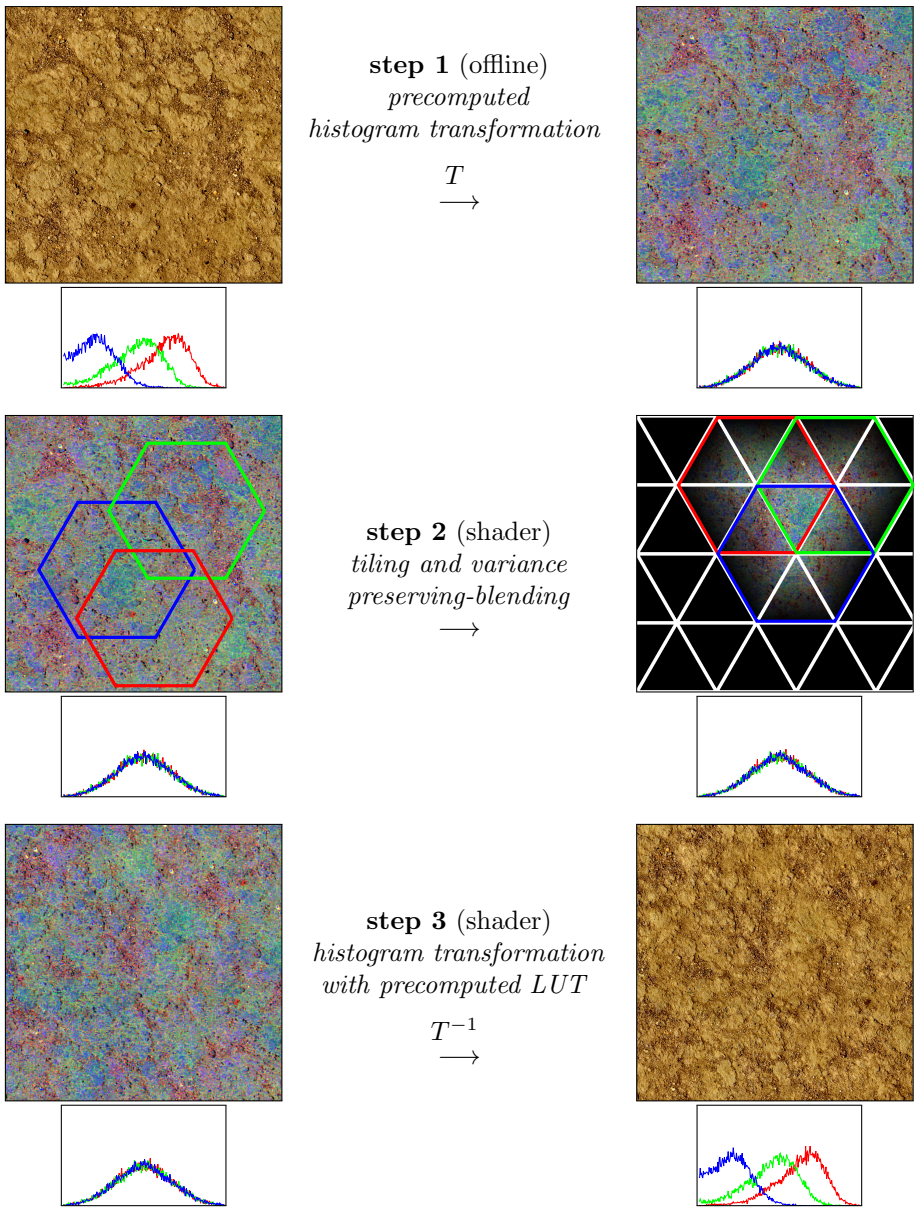

Figure 1.4. Tiling and blending with histogram-preserving blending.

1.3 Precomputing the Histogram Transformations

This section is dedicated to the C++ precomputation of the histogram transformation T applied on the input image and the inverse histogram transformation T^{-1} stored in a look-up table that are passed to the fragment shader in Listing 1.4.

1.3.1 Target Gaussian distribution

As shown in Figure 1.4, T is an histogram transformation that makes the input distributed as a Gaussian distribution $\mathcal{N}\left(\mu, \sigma^2\right)$ whose Probability Density Function (PDF) is

$$\mathrm{PDF}(x) = \frac{1}{\sqrt{2\,\pi\,\sigma^2}}\,\exp\left(-\frac{(x-\mu)^2}{2\,\sigma^2}\right). \tag{1.3}$$

To do this, we need to choose the parameters of the Gaussian distribution we will be using and recall some of its properties.

Parameters We choose the target Gaussian distribution of parameters $\mu = \frac{1}{2}$ and $\sigma^2 = \frac{1}{6^2}$. With these parameters, the distribution fits well in the interval $[0, 1]$ and can be stored with 8-bit precision.

Cumulative Distribution Function The histogram transformation T in Section 1.3.2 requires the Cumulative Distribution Function (CDF) of the Gaussian distribution. It is the function that computes the quantile values of the distribution at a given position x:

$$\mathrm{CDF}(x) = \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x-\mu}{\sigma\,\sqrt{2}}\right)\right]. \tag{1.4}$$

A quantile value $U = \mathrm{CDF}(x)$ is the integral of the distribution below x. For instance, if $U = 0.30$ it means that 30% of the integral is below x and 70% is above.

```
float CDF(float x, float mu, float sigma)
{
    float U = 0.5f * (1 + Erf((x-mu)/(sigma*sqrtf(2.0f))));
    return U;
}
```

Listing 1.6. Cumulative Distribution Function (CDF) of a Gaussian.

Inverse Cumulative Distribution Function The inverse histogram transformation T^{-1} in Section 1.3.3 requires the inverse CDF:

$$\mathrm{CDF}^{-1}(U) = \mu + \sigma\sqrt{2}\,\mathrm{erf}^{-1}(2U-1)\,. \tag{1.5}$$

It computes the quantile $x = \mathrm{CDF}^{-1}(U)$ of a given value $U \in [0,1]$.

```
float invCDF(float U, float mu, float sigma)
{
    float x = sigma*sqrtf(2.0f) * ErfInv(2.0f*U-1.0f) + mu;
    return x;
}
```

Listing 1.7. Inverse Cumulative Distribution Function (ICDF) of a Gaussian.

1.3.2 Applying the histogram transformation T on the input

In this section, we show how to apply the histogram transformation T on the input (step 1 in Figure 1.4). Our algorithm makes each color channel of the input distributed as the target Gaussian chosen in Section 1.3.1.

Algorithm A discrete 1D histogram transformation T is typically done by replacing sorted values I from the input by the same number of sorted values G from the target histogram, as shown in Figure 1.5.

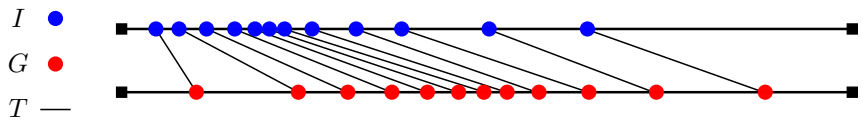

Figure 1.5. Histogram transformation of the input. *We sort the pixel values I of the input and we map them to sorted values G from the target Gaussian distribution.*

Implementation In Listing 1.8, we start by sorting the values of the input image. For this purpose, we use a structure **PixelSortStruct** that stores the coordinates and the value of a pixel. Then, we go through the sorted list of pixel values and for the i-th element we compute its quantile value $U = \frac{i+\frac{1}{2}}{N}$. It means that $U\%$ of the list is before this element and $(1-U)\%$ is after. We replace the pixel value by the same quantile in the Gaussian distribution using the inverse CDF of Equation (1.5): $G = \mathrm{CDF}^{-1}(U)$.

```
void ComputeTinput(TextureDataFloat& input, TextureDataFloat& T_input,↩
     int channel)
{
    // Sort pixels of example image
    vector<PixelSortStruct> sortedInputValues;
    sortedInputValues.resize(input.width * input.height);
    for (int y = 0; y < input.height; y++)
    for (int x = 0; x < input.width; x++)
    {
        sortedInputValues[y * input.width + x].x = x;
        sortedInputValues[y * input.width + x].y = y;
        sortedInputValues[y * input.width + x].value = input.GetPixel(↩
            x, y, channel);
    }
    sort(sortedInputValues.begin(), sortedInputValues.end());

    // Assign Gaussian value to each pixel
    for (unsigned int i = 0; i < sortedInputValues.size() ; i++)
    {
        // Pixel coordinates
        int x = sortedInputValues[i].x;
        int y = sortedInputValues[i].y;
        // Input quantile (given by its order in the sorting)
        float U = (i + 0.5f) / (sortedInputValues.size());
        // Gaussian quantile
        float G = invCDF(U, GAUSSIAN_AVERAGE, GAUSSIAN_STD);
        // Store
        T_input.SetPixel(x, y, channel, G);
    }
}
```

Listing 1.8. Applying the histogram transformation T on the input.

1.3.3 Precomputing the inverse histogram transformation T^{-1}

In this section, we show how to compute the inverse histogram transformation T^{-1} that maps Gaussian values to values from the input and store it in a look-up table (step 3 in Figure 1.4).

Algorithm The algorithm consists in mapping sorted values, as in the previous section (Figure 1.5). However, the computation of the values is different. Since we use a Gaussian distribution that can be well represented in the interval $[0, 1]$, we are going to parameterize the look-up table on this interval and we associate quantiles of the Gaussian distribution in $[0, 1]$ to quantiles of the pixel values.

Implementation In Listing 1.9, we start by sorting the values of the input image. Note that an optimized implementation could reuse the sorting step of Listing 1.8. Then, we go through the texels of the look-up table that parameterizes the interval $[0, 1]$ such that the i-th over N texel is associated with the position $x = \frac{i+\frac{1}{2}}{N}$. We compute the Gaussian quantile value at this position using Equation (1.4): $U = \mathrm{CDF}(x)$, and we pick up the

same quantile in the sorted pixel values, i.e we fetch the $(U.M)$-th element in the sorted list if it has M entries. This is the value that we store in the look-up table.

```
void ComputeinvT(TextureDataFloat& input, TextureDataFloat& Tinv, int ↩
    channel)
{
    // Sort pixels of example image
    vector<float> sortedInputValues;
    sortedInputValues.resize(input.width * input.height);
    for (int y = 0; y < input.height; y++)
    for (int x = 0; x < input.width; x++)
    {
        sortedInputValues[y * input.width + x] = input.GetPixel(x, y, ↩
            channel);
    }
    sort(sortedInputValues.begin(), sortedInputValues.end());

    // Generate Tinv look-up table
    for (int i = 0; i < Tinv.width; i++)
    {
        // Gaussian value in [0, 1]
        float G = (i + 0.5f) / (Tinv.width);
        // Quantile value
        float U = CDF(G, GAUSSIAN_AVERAGE, GAUSSIAN_STD);
        // Find quantile in sorted pixel values
        int index = (int)floor(U * sortedInputValues.size());
        // Get input value
        float I = sortedInputValues[index];
        // Store in LUT
        Tinv.SetPixel(i, 0, channel, I);
    }
}
```

Listing 1.9. Precomputing the inverse histogram transformation T^{-1} and storing it in a look-up table.

1.3.4 Discussion

With the fragment shader of Section 1.2 and the precomputations of Section 1.3 we already have a standalone implementation. However, this implementation has several shortcomings: color problems might appear with some inputs. They are due to computing separate per-channel histogram transformations and the incompatibility of mipmapping and using a look-up table. Sections 1.4 and 1.5 are dedicated to overcome these shortcomings.

1.4 Improvement: using a decorrelated color space

Our method, as described so far, occasionally produces procedural texture that exhibit colors that were not present in the example texture, as in Figure 1.6(b). In this section, we show how to reduce this problem by using a decorrelated color space, such as [Heeger and Bergen 95].

1.4.1 The problem with color-space correlations

In Section 1.3, we computed histogram transformations for each color channel separately, which occasionnaly produces wrong colors in the output. Indeed, the histogram of an RGB image is not the composition of three 1D functions but rather one 3D function or a 3D point cloud, as shown in Figure 1.6(a). This 3D histogram might have inter-channel correlations and transforming the channels separately does not preserve these correlations. For instance, the result of Figure 1.6(b) has the same 1D histogram as the input for each channel. However, since the inter-channels correlations are not preserved, the 3D shape of this histogram is not preserved and wrong colors appear in the result. We obtained the result of Figure 1.6(c) by using a color space in which the channels are not correlated such that processing them separately is less prone to this problem.

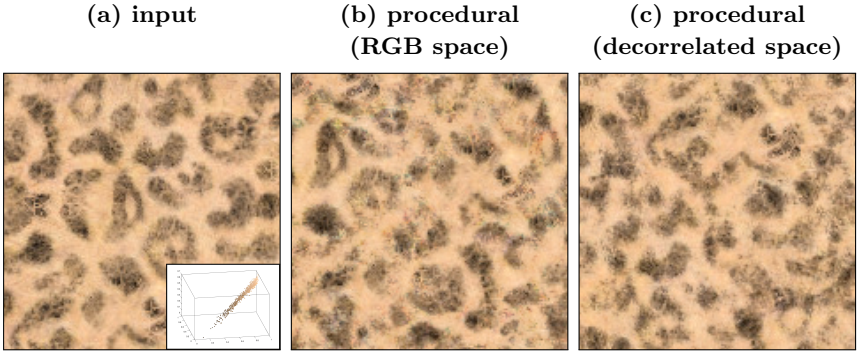

Figure 1.6. Improvement: using a decorrelated color space. *If the color channels are correlated, processing them separately might introduce wrong colors (b) that were not present in the input (a). We reduce this problem by using a color space in which the channels are not correlated (c).*

1.4.2 Decorrelating the color space

We precompute the color space transformation before step 1 in Figure 1.4) and revert it at the end of the fragment shader after step 3 in Figure 1.4.

Precomputation We start by computing the covariance matrix of the input histogram and extracting its eigenvectors, which means extracting the principal axes of the point cloud given by the pixel's colors in the RGB space, as shown in Figure 1.7(a). Along these eigenvectors, the coordinates of the points are statistically decorrelated. Then, we compute the bounding box of the point cloud aligned with these eigenvectors and we find the coordinates of the points in this bounding box, as shown in Figure 1.7(b). With this parameterization, all the points are parameterized by

$$\mathbf{P} = \mathbf{O} + v_1\,\mathbf{V}_1 + v_2\,\mathbf{V}_2 + v_3\,\mathbf{V}_3 \text{ with } (v_1, v_2, v_3) \in [0,1]^3, \tag{1.6}$$

where the bounding box is defined by its corner $\mathbf{O}$ and its orthogonal axes $\mathbf{V}_1$, $\mathbf{V}_2$ and $\mathbf{V}_3$.

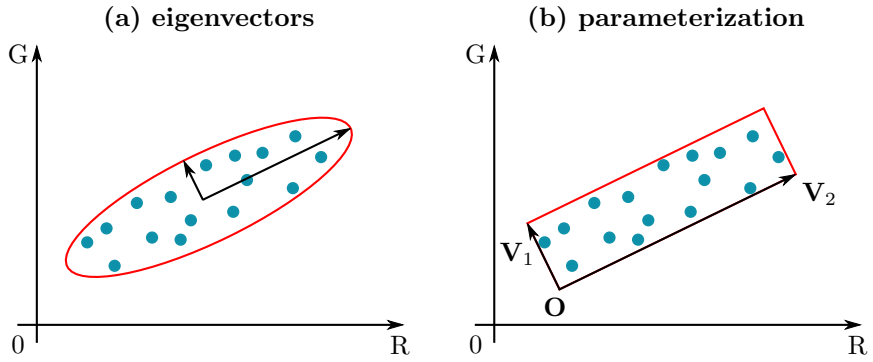

Figure 1.7. Parameterization of the decorrelated color space.

Fragment shader Before returning, the fragment shader transforms the result back to the original color space with Equation (1.6). This is done in the function **ReturnToOriginalColorSpace** provided in Listing 1.10.

```
uniform vec3 colorSpaceVector1;
uniform vec3 colorSpaceVector2;
uniform vec3 colorSpaceVector3;
uniform vec3 colorSpaceOrigin;

vec3 ReturnToOriginalColorSpace(vec3 color)
{
    vec3 result =
        colorSpaceOrigin +
        colorSpaceVector1 * color.r +
        colorSpaceVector2 * color.g +
        colorSpaceVector3 * color.b;
    return result;
}
```

Listing 1.10. Return to the original color space in the fragment shader.

1.5 Improvement: prefiltering the look-up table

Our method, as described so far, uses mipmap levels to fetch the Gaussian input in step 2 of Figure 1.4. However, when the lower levels of detail are used, comparing its appearance to a regular tiling of the input reveals an issue of colour deviation, as shown in Figure 1.8. In this section, we show how to solve this problem by prefiltering the look-up table.

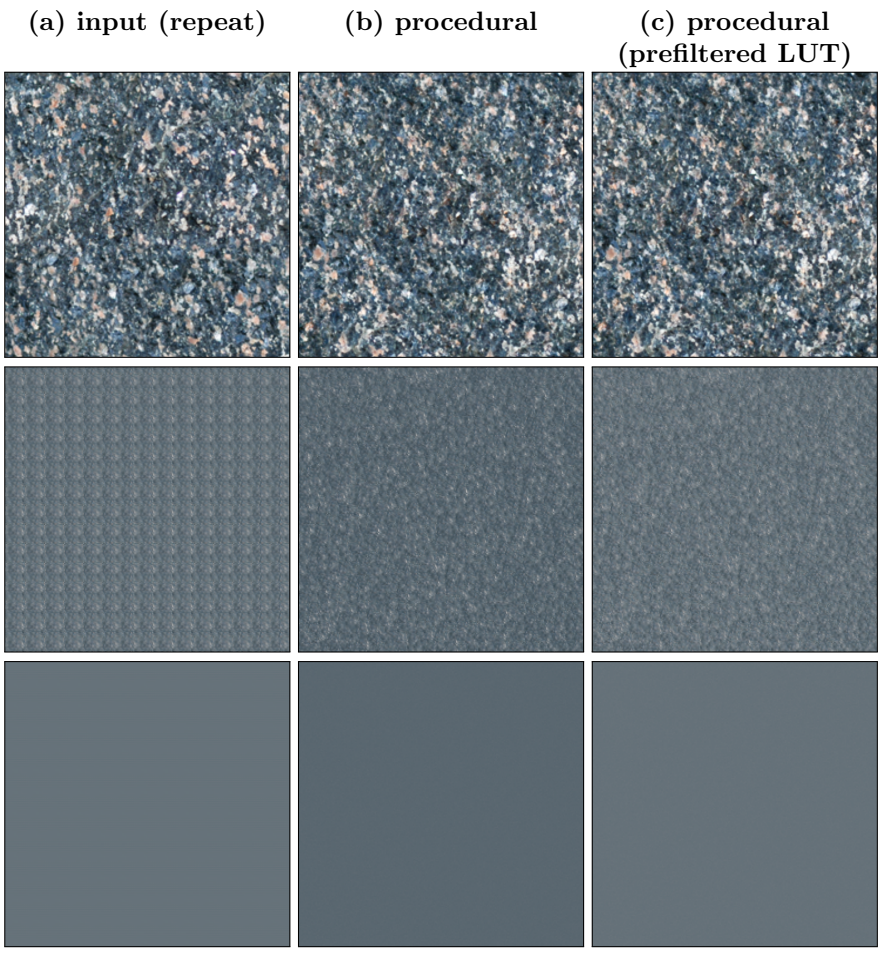

Figure 1.8. Improvement: prefiltering the look-up table. *The procedural texture uses a look-up table on top of the mipmapped input. This results in a noticable color shift as we zoom out (b) compared to the input (a). We solve this problem by prefiltering the look-up table (c).*

1.5.1 The problem with texture filtering and look-up tables

Classic texture filtering To understand the problem when filtering our procedural texture, we look into the equation of classic texture filtering. We define **texture**(uv) as the color of the input texture at a position uv and P the domain covered by the pixel footprint. Figure 1.9 illustrates that the filtered color is the integral of the texture over the pixel footprint:

$$\textbf{filtered texture} = \int_P \textbf{texture}(uv)\,\mathrm{d}uv. \tag{1.7}$$

Texture mipmapping (with anisotropic filtering for more accuracy) provides a fast way to evaluate this integral.

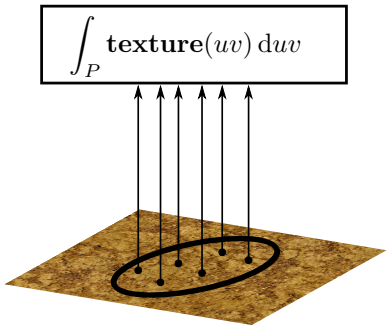

Figure 1.9. Classic texture filtering.

Procedural texture filtering (reference) Our tiling-and-blending method computes a procedural texture that is the composition of the Gaussian input texture and a look-up table (LUT) that contains the inverse histogram transformation:

$$\textbf{procedural texture}(uv) = \textbf{LUT}\left(\textbf{texture}(uv)\right). \tag{1.8}$$

If we apply Equation (1.7) to this formulation, we obtain the following filtering equation:

$$\textbf{filtered procedural texture} = \int_P \textbf{LUT}\left(\textbf{texture}(uv)\right)\,\mathrm{d}uv. \tag{1.9}$$

As shown in Figure 1.10, this integral can be computed by sampling the values of the texture over the footprint P, passing the values through the look-up table, and averaging the results. Unfortunately, this process is too costly and we are thus willing to use mipmapping, as for a conventional texture.

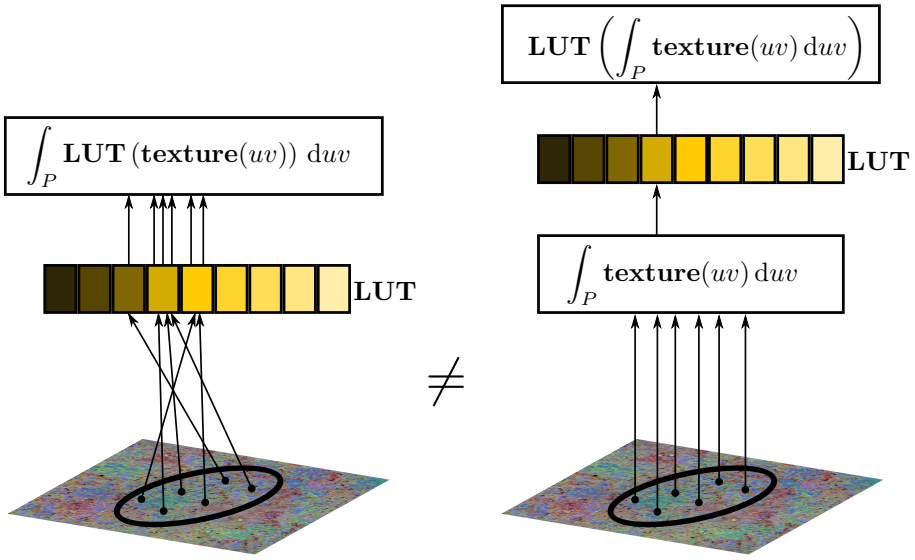

Figure 1.10. Filtering the procedural texture. *The correct filtering averages the values after the application of the look-up table. Filtering the texture before and applying the look-up table after does not produce the same result.*

Filtering the procedural texture (wrong) A simple approach consists in using a mipmapped version of the input texture, fetching a single sample from it as for a conventional texture, and then passing it through the look-up table, as shown in Figure 1.10. However, this computes

$$\textbf{filtered procedural texture} = \mathbf{LUT}\left(\int_P \mathbf{texture}(uv)\,\mathrm{d}uv\right), \quad (1.10)$$

which is not the right result because the integral and the look-up table do not commute:

$$\mathbf{LUT}\left(\int_P \mathbf{texture}(uv)\,\mathrm{d}uv\right) \neq \int_P \mathbf{LUT}\,(\mathbf{texture}(uv))\;\mathrm{d}uv. \quad (1.11)$$

This inequality explains the color difference between Figure 1.8(a) and (b).

1.5.2 Alternative filtering formulation with a look-up table

We use the solution of [Heitz et al. 13] to the problem of filtering procedural textures with look-up tables (also called "color maps"). Their solution is based on the observation that the reference result of Equation (1.9) is a

weighted average of values from the look-up table. Hence, the equation can be rewritten

$$\textbf{filtered procedural texture} = \int_{-\infty}^{+\infty} \mathbf{LUT}(t)\, H_P(t)\, \mathrm{d}t, \qquad (1.12)$$

where H_P gives the weight of each entry of the look-up table. This weight depends on the distribution of texture values t inside the pixel footprint P. The more a value t of the texture is represented, the more the entry $\mathbf{LUT}(t)$ contributes to the weighted average. Hence, H_P is the histogram of the values of the texture inside P. This equivalence is shown in Figure 1.11.

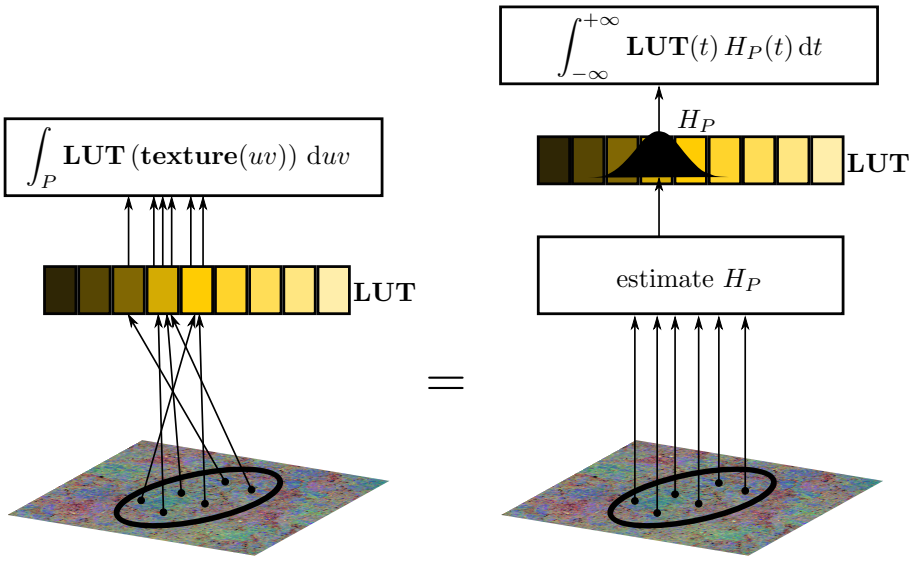

Figure 1.11. Alternative filtering formulation with a look-up table. *Filtering the texture with the look-up table is equivalent to convolving the look-up table by the histogram of the texture values inside the pixel footprint.*

Implementation with a prefiltered look-up table Applying the result of Equation (1.12) in practice requires estimating H_P for a given footprint P and computing its product integral with the look-up table. To do this in real-time, we approximate H_P by a Gaussian distribution and use a look-up table prefiltered with a Gaussian filter for each level of detail of the input texture. The motivation for this approximation is that at the texture has effectively a Gaussian histogram. Hence, the approximation becomes

exact at the highest level of detail and remains reasonable at intermediate levels.

1.5.3 Computing and fetching the prefiltered look-up table

Precomputation In our implementation, we prefilter the look-up table in a function **PrefilterLUT**. This function creates a 2D look-up table whose width is the same as the unfiltered look-up table and whose height is the number of levels of detail of the input texture. For each level of detail L we compute the average variance in all the subwindows of width 2^L. At the first level of detail the variance is 0 and at the highest level of detail the variance is the variance of the full Gaussian texture, which is $\frac{1}{6^2}$ as explained in Section 1.3.1. For each level of detail, we filter the look-up table by a Gaussian filter of the associated variance.

Fragment shader We update the fragment shader in Listing 1.11 where we use the function **textureQueryLod** to obtain the level of detail of the input texture and we remap it to a value in $[0, 1]$ to obtain a y coordinate to fetch the look-up table.

```
// Compute LOD level to fetch the prefiltered look-up table invT
float LOD = textureQueryLod(Tinput, uv).y / float(textureSize(invT↩
    , 0).y);

// Fetch prefiltered LUT (T^{-1})
vec3 color;
color.r = texture(invT, vec2(G.r, LOD)).r;
color.g = texture(invT, vec2(G.g, LOD)).g;
color.b = texture(invT, vec2(G.b, LOD)).b;
```

Listing 1.11. Fetching the prefiltered look-up table in the fragment shader.

1.6 Improvement: using compressed texture formats

In Figure 1.12, we test our algorithm with the DXT1 compressed texture format applied to the Gaussian version of the input **Tinput** and the look-up table **invT**. We notice that the compression occasionally introduces visible artifact when it is applied directly on our textures (Figure 1.12(b)) and a modification is necessary to support a compressed texture format. The problem is that our histogram transformation makes all the channels have the same range of Gaussian values. This impacts the quality of the compression because the compressor optimizes an error that has become equally distributed among the channels while the true error should be more important for channels with wide ranges. Fixing this issue is simple: instead of using the same Gaussian distribution $\mathcal{N}\left(\frac{1}{2}, \frac{1}{6^2}\right)$ for all the channels, we scale the Gaussian distribution such that its standard deviation around the average $\frac{1}{2}$ becomes proportional to the actual range of the channel data. We do this modification just before sending the data to the DXT compressor and we revert it in the fragment shader. With this minor modification we were able to fix the issue and safely use the DXT1 format for all our textures (Figure 1.12(c)). Our C++ OpenGL demo provides a binary flag #define USE_DXT_COMPRESSION that enables these modifications.

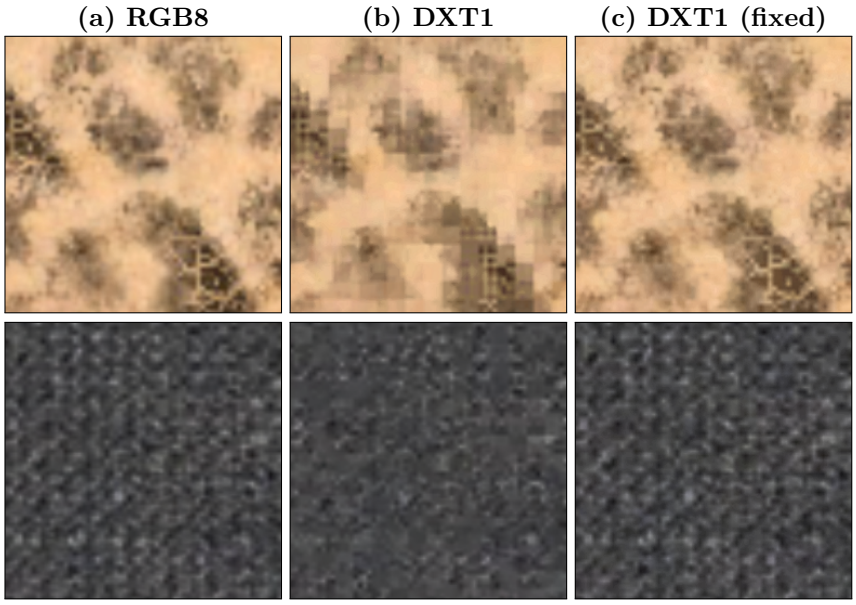

Figure 1.12. Using a compressed texture format. *The DXT1 texture format fails with some inputs if it is applied directly on the textures compute by our algorithm (b). We fix this problem by scaling the range of the Gaussian texture (c).*

1.7 Results

Performance and storage In Table 1.1, we compare the performance and storage of our method compared to a classic texture repeat, as in Figures 1.13 and 1.14. On average, it is 4-5 times costlier, which makes sense since we fetch the input 3 times, use one additional look-up table fetch and use a few additional operations. The repeated tiling only requires the storage of the input texture while our method requires the storage of the Gaussian input (Tinput) and the look-up table (Tinv). Since the Gaussian input has the same size as the input, the memory overhead of our method is only the storage of the look-up table, which is small in comparison.

size	**format**	**(T)input**	**invT**	**repeat**	**procedural**
64^2	RGB8	16KB	2KB	0.035ms	0.179ms
128^2	RGB8	65KB	3KB	0.035ms	0.180ms
256^2	RGB8	262KB	3KB	0.036ms	0.181ms
512^2	RGB8	1048KB	3KB	0.039ms	0.186ms
1024^2	RGB8	4194KB	4KB	0.052ms	0.200ms
2048^2	RGB8	16777KB	5KB	0.112ms	0.341ms
64^2	DXT1	3KB	$\leq$1KB	0.035ms	0.180ms
128^2	DXT1	11KB	$\leq$1KB	0.035ms	0.180ms
256^2	DXT1	48KB	$\leq$1KB	0.035ms	0.180ms
512^2	DXT1	174KB	$\leq$1KB	0.036ms	0.180ms
1024^2	DXT1	699KB	$\leq$1KB	0.039ms	0.182ms
2048^2	DXT1	2796KB	$\leq$1KB	0.046ms	0.207ms

Table 1.1. Performance and storage comparison. *We compare our method to a single texture fetch in a repeated texture for various sizes of the input texture and storage formats. The classic repeat requires only the storage of the* **input** *texture. Our method requires the storage of the Gaussian version of the input* **Tinput**, *which has the same size as the input, and the look-up table* **invT**. *We measured the performance by rendering a full-screen quad at* 1920×1080 *resolution on a GeForce GTX 980.*

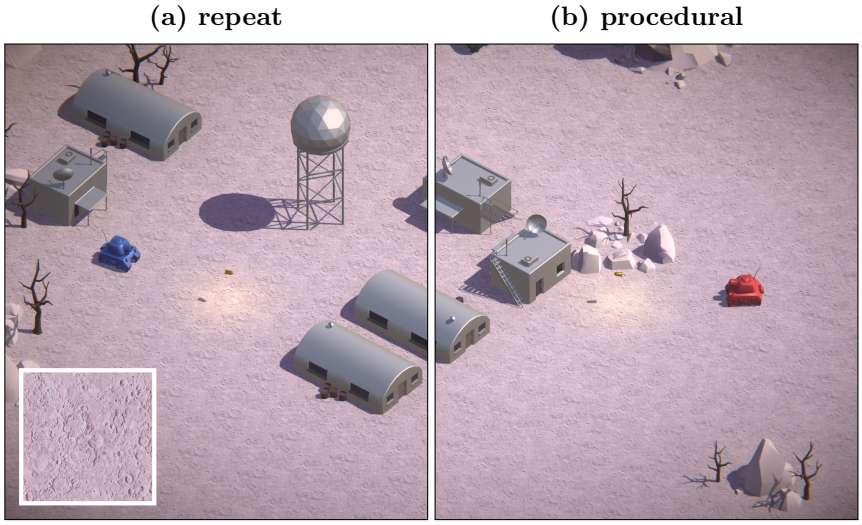

Figure 1.13. Comparison of classic texture repeat and our procedural texturing algorithm applied on the ground texture of a video game scene.

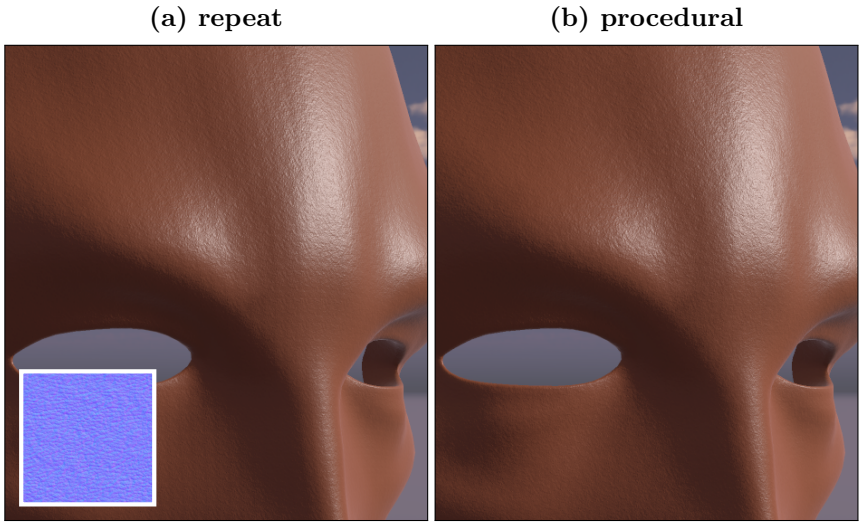

Figure 1.14. Our algorithm applied on non-RGB input. *We compare classic texture repeat and our procedural texturing algorithm on a small-scale skin pore normal map.*

Generative textural space Our method is dedicated to stochastic textures, such as the rock in Figure 1.15. It does not produce plausible results if the input presents a strong pattern-like organization like in Figure 1.16.

Figure 1.15. Our procedural texturing algorithm applied on a rock texture.

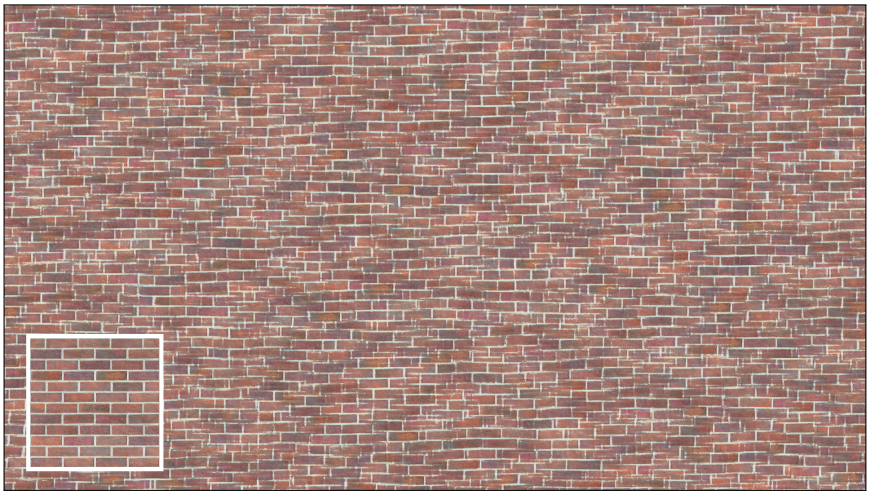

Figure 1.16. Failure case of our method. *Our method does not produce plausible results if the input presents a strong pattern-like organization.*

1.8 Conclusion

We have presented an implementation of our procedural texturing algorithm that works well for breaking the repetition of tiled textures. This algorithm is meant to be used with stochastic textures (moss, granite, sand, etc.) and cannot be used with repetitive or strongly correlated patterns. It has little memory overhead, works well with the compressed DXT texture format, and is about four times the cost of a classic texture repeated tiling. Finally, it is straightforward to adapt it to other inputs than RGB color data such as the normal map in Figure 1.14.

1.9 Acknowledgments

This chapter is the result of Thomas Deliot's master thesis, which was supervised by Eric Heitz. Both authors conducted this work at Unity Technologies.

Bibliography

[Heeger and Bergen 95] David J. Heeger and James R. Bergen. "Pyramid-based Texture Analysis/Synthesis." In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, *SIGGRAPH '95*, pp. 229–238, 1995.

[Heitz and Neyret 18] Eric Heitz and Fabrice Neyret. "High-Performance By-Example Noise using a Histogram-Preserving Blending Operator." *Proceedings of the ACM on Computer Graphics and Interactive Techniques (issue for the ACM SIGGRAPH / Eurographics Symposium on High-Performance Graphics 2018)* 1:2 (2018), 25.

[Heitz et al. 13] Eric Heitz, Derek Nowrouzezahrai, Pierre Poulin, and Fabrice Neyret. "Filtering Color Mapped Textures and Surfaces." In *I3D'13 - ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, *I3D '13 Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 129–136. ACM, 2013.

[Perlin 01] Ken Perlin. "Noise hardware.", 2001. Real-time shading languages, SIGGRAPH 2001 Course.